Flowtables: Program Skeletal Inversion for Defeat of Interprocedural Analysis with Unique Metamorphism*

Luke Jones, Jeremy Blackthorne, Ryan Whelan, Graham Baker
MIT Lincoln Laboratory
{luke.jones, jeremy.blackthorne, rwhelan, gzbaker}@ll.mit.edu

ABSTRACT

Obfuscation, in the most general sense, is widely applicable to intellectual property protection, software tamper resistance and cryptographic algorithms. We have created Flowtables, a LLVM-based obfuscator which aims to protect intellectual property, hardening programs against analysis by relocating the edges of the call graph to a different process. This process temporarily and minimally supplies edges back to the original program only at runtime. We call this transformation *program skeletal inversion*, and by effectively removing the call graph from a program, we defeat any interprocedural analyses. In addition, the newly externally malleable program skeleton enables unique metamorphism, with the beneficial property of arbitrarily complex functionality transformation.

1. INTRODUCTION

Flowtables (FT), our C++/LLVM prototype for novel obfuscating transformations, grants practical enhancements to intellectual property security by applying a simple primitive to a complex part of a program. Namely, our tool can remove the edges of a program's call graph and locate them elsewhere. Naturally, the call graph of a program cannot be permanently removed like comments and symbol names can, so definable subsets of call graph edges are supplied back to the obfuscated program as needed at runtime. This exposure of the call graph to external manipulation, or *program skeletal inversion* (PSI), in addition to enabling the removal of a resident call graph, also supplies the basis for unique metamorphic potential ranging from changing the entire behavior of a program in an instant, to maintaining functional equivalence but syntactic divergence by interchanging call graph edges with edges to cloned functions. Our assertions of security derive first from an analysis of the implications of call graph removal from a traditional program analysis perspective and then from a consideration of the most probable adversaries of our scheme.

This paper proceeds by first historically introducing obfuscation, its evolution, and FT's place in the evolution. Next, we consider our approach abstractly, then we discuss the actual implementation of FT, followed by evaluation of the theoretical security and efficiency of PSI transformed programs. Then, we examine the empirical results of correctness, completeness, and performance. Finally, we close with related work, future work and conclusions.

2. BACKGROUND

Some of the earliest recorded examples of obfuscation are the results of the International Obfuscated C Code Contest (IOCCC) [20] starting in 1984. For naught but hacker credibility and enjoyment, C programmers created thoroughly obtuse source code files that did indiscernible, unexpected, but well-defined things. Other programmers reverse engineered them to determine who created the most ingenious obfuscated program. These obfuscation creations represented the birth of a hobby in popular computer scientist culture, and proceeded in a mostly manual process.

Authors of unwanted software first developed automatic obfuscation, which was necessary for survival in hostile environments. These authors first explored obfuscation with the Brain virus in 1986 which used "garden-pathing" to cover up the modifications that it made to the disk without permission [22]. However, malicious code authors started code protection via obfuscation in earnest when they started creating self-decrypting executables such as Cascade [3], which led to the entire arms race that may be summed up in a singular word: morphism [28]. After anti-virus programmers learned to look for decrpytor stubs in encrypted malware, oligomorphism developed which gave malware multiple decryptor stubs to select. Once anti-virus vendors learned to detect this too, malware authors developed polymorphism which in most cases further defended the decryptor stubs, but left the actual payload or code bodies the same after decryption and thus unprotected. Anti-virus authors parried malicious polymorphism with emulation. With emulation, their scanners would wait for malware to come to rest in its final state and then search for signatures. Finally, metamorphic viruses surfaced which mutated their own code bodies through obfuscation primitives like dead code insertion and code permutation [26]. One of the first metamorphic samples was the Regswap virus, found in 1998, which avoided

*Distribution A: Public Release. This work is sponsored by the Department of the Air Force under Air Force Contract #FA8702-15-D-0001. Opin-ions, interpretations, conclusions and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

detection by switching the registers that its instructions used [22].

In 1997, Some fifteen years after the start of the IOCCC, and congruent with the last phase of the morphism arms race, a road map for practical obfuscation appeared in computer science academics with Collberg's seminal work on obfuscation taxonomy [9]. Next in 2001, Barak et al. established the groundwork for theoretical obfuscation in [2]. Future obfuscation researchers would often be guided by the frameworks established in these two papers.

We present the obfuscation research most relevant to our work in the progress of the past decade and a half in Section 6. Our approach continues the progression towards the most efficacious blend of theory and practice.

3. APPROACH

The key features of FT are the program skeletal inversion (PSI) algorithms. We apply PSI to the target of obfuscation, $\mathcal{P}$. Through program transformation, next we supply the resulting analysis information to the external controller, $\mathcal{C}$. At runtime the controller can induce metamorphism in the target program for several purposes, such as execution with only partial, time-sliced call graph disclosure (Section 3.2) or bundling multiple code modules or projects into the same binary (Section 3.3.2).

The transformation of a program $\mathcal{P}$ to $\mathcal{P}''$ with these algorithms should maintain the following invariants. $\mathcal{P}'$ is simply an intermediary form of $\mathcal{P}''$:

- $\mathcal{P}''$ should incur no more than a polynomial space increase over $\mathcal{P}$, either statically or at runtime. Appendix A shows that we meet this requirement.
- $\mathcal{P}''$ should incur no more than a polynomial execution time increase over $\mathcal{P}$. We demonstrate this in Section 5.2.
- $\mathcal{P}''$ must be represented as its original functions stripped of any call graph edges, but including a table mediating new call graph edges, and any code required to allow the program to be generated and run in this state. We demonstrate this in the subsequent discussion.

A generated instance of $\mathcal{P}''$ must have the following properties:

- The table portion of $\mathcal{P}''$ must be externally malleable
- $\mathcal{P}''$ must operate correctly if the table is externally modified to another correct state

We refer to the aforementioned table as the *flowtable* and symbolically represent it with Φ. For convenience, we use Φ in three distinct ways. Absent any other notation, Φ represents the table containing all correct call graph edges. Used in conjunction with set notation, $\Phi\{\varphi_0, \varphi_1, ..., \varphi_n\}$ is the set of all possible table configurations. Lastly, Φ[y] or φ[x] used with array notation represents a correct call graph edge "y" or an edge in a particular flowtable instantiation of $\mathcal{P}''$, "x". In addition to the *flowtable*, we define the possible partitions of the call graph as the set Π, the set of callsites within partitions as Ξ, and the set of possible *flowtable* configurations as Φ.

The transformation of $\mathcal{P}$ to $\mathcal{P}''$ is accomplished using the following three algorithms:

Partitioning analysis - This analysis pass partitions functions in the call graph into sets of distinct subgraphs, $\pi_i \in \Pi$, according to a max number of functions per partition, n_p. It also keeps track of the sets of call sites in each partition, $\xi_i \in \Xi$ for use in the flow indirection transformation.

Gating transformation - This transformation pass inserts gate code at the beginning of function prologues and end of function epilogues in any function in the upper fringe of all partitions. Upper fringe functions are defined in Section 3.1.1. The gate code self-reports what partition is needed and waits until the controller provides an update.

Flow indirection - This transformation pass changes every applicable call instruction to use the appropriate flowtable index instead of its original function pointer.

In Figures 1,2 and 3, for simplicity, we assume a partition size $n_p = 1$ so that every function gets its own partition and gate. These partitions are used for $\mathcal{P}''$ to communicate with a controller program, $\mathcal{C}$, called *Flowcontrol*, so that $\mathcal{P}''$ never dereferences an invalid pointer in φ[ ]. The GATE function sends the current partition needed, π_i, to $\mathcal{C}$ and blocks while waiting for φ_i which contains all the call graph edges that can possibly be traversed by $\mathcal{P}''$ while still in partition π_i. For clarity, we omit the exit gates at the end of each function in Figure 1, but these serve to make $\mathcal{P}''$ communicate when it is traversing its call graph in reverse through return instructions so that the correct φ[ ] can be maintained.

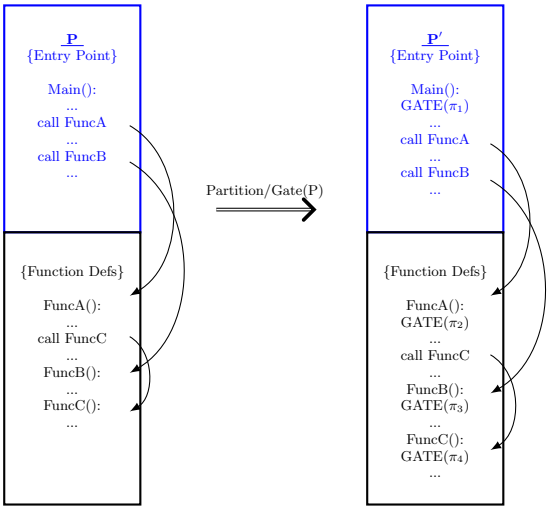


Figure 1: Partitioning and gating $\mathcal{P}$

Calls to functions are directed through the *flowtable*, as demonstrated by Figure 2. The transformation creates a master flowtable, Φ, for use by $\mathcal{C}$ to determine minimally correct φ_i for each π_i self-reported by $\mathcal{P}''$ at runtime. The initial flowtable resident in the final form of $\mathcal{P}''$, φ_0, has no correct entries. The entries can either be blank or incorrect.

With φ_0 being completely incorrect, $\mathcal{P}''$ now requires $\mathcal{C}$ to function properly as shown in Figure 3. As soon as $\mathcal{P}''$ begins executing, it reaches GATE(π_1) which tells $\mathcal{C}$ that $\mathcal{P}''$ is about to enter partition π_1. $\mathcal{C}$ looks up the correct

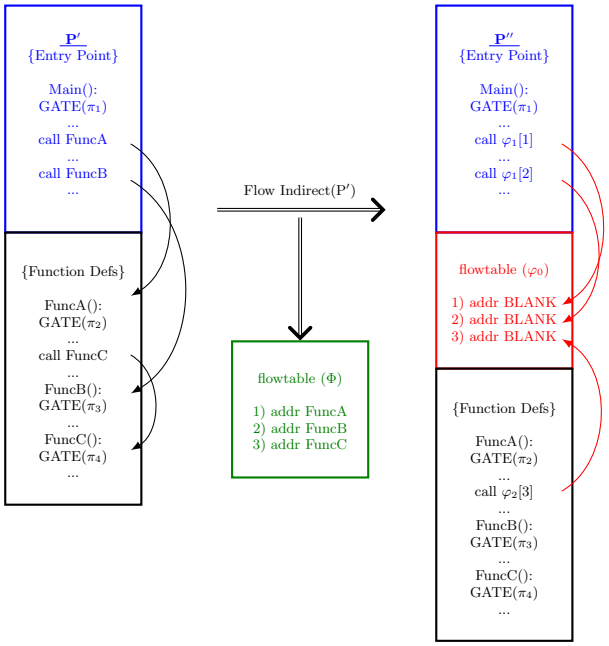


Figure 2: Directing calls through the flowtable

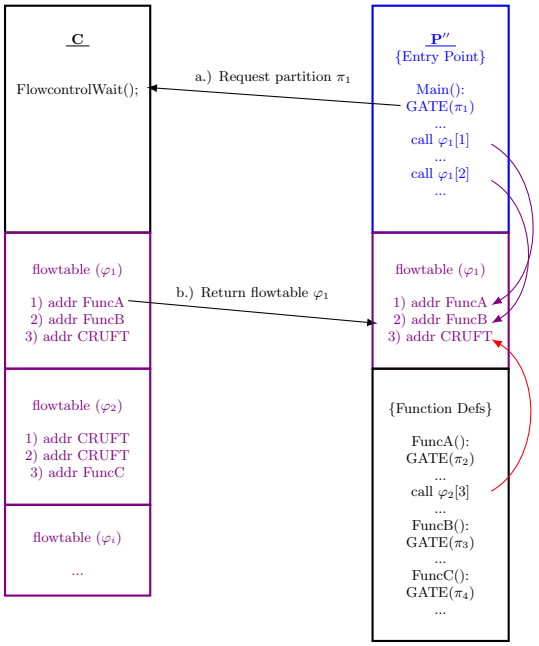


Figure 3: $\mathcal{P}''$ and $\mathcal{C}$ communicate so that $\mathcal{P}''$ runs normally with a relocated call graph

flowtable configuration for π_i, φ_i, and sends it to $\mathcal{P}''$. Now, with a minimally correct $\varphi[\,]$, $\mathcal{P}''$ resumes execution. Soon it calls $\varphi_1[1]$ which dereferences to FuncA. If we continue in FuncA, then $\mathcal{P}''$ hits GATE(π_2) and requests π_2 from $\mathcal{C}$ which supplies φ_2 which has correct entries for $\varphi_2[3]$, but otherwise invalid call graph edges. Though not pictured, once $\mathcal{P}''$ returns from FuncA, it will hit the epilogue gate and request the last table configuration it had before φ_2, so $\mathcal{C}$ will supply back φ_1 and $\mathcal{P}''$ will properly call FuncB at $\varphi_1[2]$.

3.1 Program Skeletal Inversion Algorithms and Invariants

Now after looking at the high level graphical overview of Flowtables, we will look at the algorithms used to accomplish each analysis or transformation.

3.1.1 Partitioning Algorithm

Algorithm 1 is applied to $\mathcal{P}$ in order to establish a knob for adjusting the balance of security and efficiency in any given obfuscation target. Having $n_p = 1$, or a separate partition for every function provides maximum security, but minimum efficiency. Having $n_p = |CG_{\mathcal{P}}|$ provides maximum efficiency, but minimal security. The algorithm first partitions functions from a random starting point in a breadth-first traversal of the call graph, keeping n_p as a maximum partition size. After partitioning, it traverses every partition and annotates any functions that are on the upper fringe of the partition; i.e., functions that have parent call graph nodes that are in a different partition, or functions are an entry point. Simultaneously, the algorithm builds a set of call sites that exist in each partition, $\xi_i \in \Xi$, for later analyses.

Inserting randomness into the partitioning process with RANDOMROTATE ensures that the partitions are not predictable, and therefore are more defensible. This is considered in the adversarial discussion in Section 5.

Algorithm 1 Partition call graph of $\mathcal{P}$ into $\pi_i \in \Pi$

Require: n_p : integer
Require: $\mathcal{BFT}[\,]$: f {functions from breadth-first traversal}
Require: UPPERFRINGE(f) : (f, true) if f is on upper fringe otherwise (f, false)
Require: CALLSITES(f) : all the call sites in f
Require: RANDOMROTATE($a[\,]$) : circularly shifts the elements of array a randomly
{main logic for partitioning}
RANDOMROTATE($\mathcal{BFT}$)
i $\Leftarrow$ 1
j $\Leftarrow$ 0
for all f in $\mathcal{BFT}$ **do**
  if j >= n_p **then**
    i $\Leftarrow$ i + 1
    j $\Leftarrow$ 0
  end if
  $\pi_i \Leftarrow f$
  j $\Leftarrow$ j + 1
end for
for all π_i in Π **do**
  for all f in π_i **do**
    replace f with UPPERFRINGE(f)
    $\xi_i \Leftarrow$ CALLSITES(f)
  end for
end for
return $\pi_i \in \Pi$ {Partitions of $CG_{\mathcal{P}}$ with annotations}
return $\xi_i \in \Xi$ {Call sites associated with each $\pi_i \in \Pi$}

3.1.2 Gating Algorithm

Algorithm 2 traverses all $\pi_j \in \Pi$ made by Algorithm 1, and inserts the partition reporting and the $\mathcal{C}$ synchronization code at every possible entry point (the upper fringe functions) of each π_j, effectively *gating* every partition. GATE blocks $\mathcal{P}''$ while waiting for $\mathcal{C}$ to respond to π_j with φ_j.

Algorithm 2 Gating $\pi_j \in \Pi$

Require: Π : partitions of $\mathcal{P}$
Require: GATE(f, π_j) : send π_j to $\mathcal{C}$ and block while waiting for φ_j from $\mathcal{C}$
{If function is in the upper fringe of a partition as determined by Algorithm 1, insert gating logic}
1: **for all** π_j in Π **do**
2: **for all** (f, bool) in π_j **do**
3: **if** bool = true **then**
4: insert GATE(f, π_j)
5: **end if**
6: **end for**
7: **end for**
8: **return** $\mathcal{P}'$

3.1.3 Flow Indirection Algorithm

Algorithm 3 uses results from the analyses performed by Algorithms 1 and 2 to finish transforming $\mathcal{P}'$ into its obfuscated state, $\mathcal{P}''$. The algorithm first traverses all the call sites recorded in ξ_i for each partition π_i, and directs each through the flowtable $\varphi_i[\ ]$, thus applying the flow transformation while building each φ_i: the minimally correct flowtable configuration for each π_i. Lastly, the algorithm clears the flowtable and records that configuration as the initial configuration, φ_0.

Algorithm 3 Flow transform $\mathcal{CS}$ in $\mathcal{P}''$

Require: Ξ : sets of call sites for $\pi_i \in \Pi$
Require: $\Phi[\ ]$: addr $[\ \sum_{\xi_i \in \Xi} |\xi_i|\ ]$ {We make $\Phi[\ ]$ the size of the sum of the number of call sites in each π_i}
Require: TARGET(c) : target function of call site c
Require: ADDR(x) : address of object x
1: k $\Leftarrow$ 1
2: **for all** $\xi_i \in \Xi$ **do**
3: **for all** c in ξ_i **do**
4: $f \Leftarrow$ TARGET(c)
5: $\varphi_i[k] \Leftarrow$ ADDR(f)
6: TARGET(c) $\Leftarrow$ ADDR($\Phi[k]$)
7: k $\Leftarrow$ k + 1
8: **end for**
9: **end for**
10: $\phi_0[1, ..., k] \Leftarrow$ CRUFT()
11: **return** $\varphi_1, \varphi_2, ..., \varphi_i \in \Phi$
12: **return** $\mathcal{P}''$

3.2 External Metamorphic Algorithm for Normal Execution

Once a program has been exposed and analyzed as described in Section 3.1, the program's newly metamorphic nature can be manipulated at runtime such that it behaves exactly as it did before, except now with a partial call graph instead of a full, resident call graph. This section details schemes for systematic, sound flowtable transformations on a program $\mathcal{P}''$.

Program transformations at runtime are limited to changes to the call graph edge list embodied in the flowtable. Let transformations occur as transitions over the domain of valid flowtable configurations as reported by Algorithm 4 and as requested by $\mathcal{P}''$. $\mathcal{C}$ simply waits for a partition request, π_j from $\mathcal{P}''$ and and then returns the correct flowtable configuration, φ_j.

Algorithm 4 Leveraging metamorphism to induce original behavior at runtime

Require: $\varphi_1, \varphi_2, ..., \varphi_i \in \Phi$
Require: GATEWAIT() : waits for $\mathcal{P}''$ to send partition request, π
Require: SEND(φ) : send $\mathcal{P}''$ flowtable configuration, φ_j
Require: CRUFT() : random addresses for misdirection
Require: END() : is $\mathcal{P}''$ or $\mathcal{C}$ terminated?
{*noise* indicates artifacts in φ_j left by CRUFT()}
1: **while not** END() **do**
2: $\pi_j \Leftarrow$ GATEWAIT()
3: $\varphi_{j+noise} \Leftarrow$ CRUFT()
4: $\varphi_{j+noise} \Leftarrow \varphi_j$
5: SEND(φ_j)
6: **end while**

Programs transformed in the aforementioned ways have the following beneficial properties:

Unique metamorphism - The usual sign of metamorphism, self-modifying code, is absent.

Arbitrarily complex metamorphism - These programs are metamorphmic ranging from allowing completely disparate programs to be bundled together and arbitrarily ran, to allowing function duplicates to be substituted for each other.

Live-observation only - These programs exhibit partial, time-sliced call graph correctness which forces reverse engineers to infer calls through observation. The only way to gain any interprocedural insight is to observe calls taken during instances of $\mathcal{P}''$ and $\mathcal{C}$ working simultaneously.

3.3 Auxiliary Transformations

We build on the PSI algorithmic foundation with the following two transformations to demonstrate further defenses for PSI and possibilities for PSI-enabled metamorphism.

3.3.1 Flow Pointer Aliasing

This auxiliary transformation provides a first step at thwarting an adversary as considered in Section 5.1.4. This transformation very simply duplicates any functions in the code body and provides the original and all of its duplicates as possible substitutions for each other in the flowtable at runtime. A live observer will only be able to trust FT entries explicitly referenced.

3.3.2 Multi-Module Linkage

This transformation combines an arbitrary number of code bodies into a single module that uses the same flowtable. When the merged program starts up, the user can choose which code body they want to execute. While not fully

integrated into an automated build process, this transformation provides all of the mechanics needed for extensions as described in Section 7.

4. IMPLEMENTATION

This section explains rationale for various platform choices, details important to the FT process but not part of the transformation algorithms, and the limitations we encountered when reifying our approach into code.

4.1 Toolchain

LLVM was chosen for building FT because of the extensiveness of functionality and documentation of its reusable libraries, as well as its well-documented and powerful intermediate representation (IR) [16]. There is already a plethora of useful tools built using LLVM [4, 11, 12] and building FT on LLVM IR ensures its interoperability with all of those tools and future LLVM-based tools.

Windows was chosen as our operating system simply because of its prevalence. Llvm-flowtables-tool could easily be built for any other system that LLVM can be built on, but the other tools in the toolchain would need to be ported for different operating systems.

Figure 4 shows how we transform $\mathcal{P}$ to $\mathcal{P}''$ and run it using $\mathcal{C}$. Rectangle nodes are programs, circle nodes are files, filled, purple nodes are products of the toolchain, and blue outlined nodes are parts of the Flowtables-suite.

4.1.1 Llvm-flowtables-tool

Llvm-flowtables-tool implements Algorithms 1, 2, and 3. Additionally it transforms $\mathcal{P}''$ as described in Sections 3.3.1 and 3.3.2. In our implementation, we provide the option to make the partitioning deterministic or pseudorandom.

4.1.2 Flowinfo

Flowinfo uses the Windows Debug Interface Access (DIA) API to gather relative virtual addresses (RVAs) from program databases (PDBs) produced from linking the object file produced by Clang with the Windows linker. The RVAs are communicated to Flowcontrol so that absolute virtual addresses can be communicated in $\varphi_i \in \Phi$.

4.1.3 Flowcontrol

Flowcontrol, or $\mathcal{C}$, implements Algorithm 4 and supports $\mathcal{P}''$ for the transformations described in Section 3.3.1 and Section 3.3.2. For testing purposes, the CRUFT implementation returns 0xEFFACED for any pointer in $\varphi_i[\,]$ that is not needed by π_i. This is so that if any flowtable entry is referenced that we were not expecting, the program throws a segmentation fault instead of continuing in an undefined state. Deployment implementations of CRUFT have many options to fill unneeded flowtable entries. Incorrect function pointers are a distinct possibility, though perhaps even other addresses might be more effective for obfuscation.

Our implementation of Flowcontrol is Windows specific, relying on interprocess communication (IPC) based on spin locks, and direct memory reads and writes. This eases debugging when both $\mathcal{P}''$ and $\mathcal{C}$ are on the same host. Future implementations could be extended to use network communication instead of IPC in order to fully leverage a separated call graph produced by FT.

4.2 Limitations

FT cannot transform any call site in $\mathcal{P}$ that is already indirect due to the complications that indirect calls introduce to program analysis. This leads to the following completeness limitations: FT ignores library calls, dynamically calculated calls and LLVM intrinsic calls. Skipping LLVM intrinsic calls is negligible, since it does not leak information to an attacker. Library calls account for 79% of all skipped call sites on average in our test-suite, and as such, addressing this is a part of ongoing work. Lastly, dynamically calculated calls account for many more call sites in C++ code than in C code, though these are not currently handled by Flowtables.

5. EVALUATION

In evaluating our FT implementation, we first consider the theoretical security of our PSI algorithms, then the theoretical efficiency of the obfuscation, and finally the empirical results that we found when testing our implementation.

5.1 Security Analysis

We approach the issue of security from a theoretical basis instead of an empirical one because there exists no metric with sufficient ground truth for assessing the security of an obfuscation scheme. Collberg presents widely used security metrics in [9], but they are useful for comparing different schemes, not providing guarantees of security. To truly assess the strength of obfuscation transformations, one must understand the theoretical limits of the scheme or of the adversary, otherwise the problem of assessing obfuscation security distills to a human cognition assessment problem [18]. Therefore in this section, we explore the theoretical extrema of security enabled by FT. In the following subsections, we consider first the best case of security for the PSI-suite, which is limiting an adversary to intraprocedural analysis by removing the call graph of a program. Next, we examine how a static analysis adversary might try to rebuild the call graph using brute-force and code artifacts to escape the austere limitations of intraprocedural analysis. Next, we consider an adversary who attempts to arbitrarily analyze $\mathcal{P}''$ dynamically. Lastly, we consider the most capable adversary, though also one that accepts the most limitations: the live-observation, dynamic analysis adversary.

5.1.1 Intraprocedural Analysis Limitation

Removal of information precludes analysis of it, but with something as integral as a call graph, we explore whether it can truly be removed from a program. The PSI algorithms enable the decoupling of a call graph from a program, and even its relocation to a different computer. Without a call graph, only intraprocedural analysis can be done [17, 27]. Considered from a different angle, analyzing removed information amounts to doing concurrent dataflow analysis without a correct concurrency model of the program [8]. Therefore, analyses based on any interprocedural information are broken by definition [5, 14], but even more so, whole program understanding requires interprocedural analysis so it is precluded by the removal of the call graph as well. However, though we remove call graph edges, with the next adversary we consider what artifacts related to the call graph are left behind and if they can be used to rebuild what was removed by FT.

5.1.2 Static Analysis Adversary

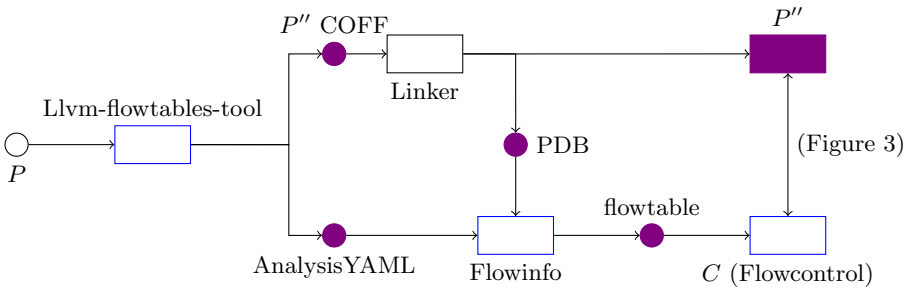


Figure 4: Applying PSI algorithms to $\mathcal{P}$ with FT toolchain

Let us first consider the adversary with the weakest capability and the strongest goal. This adversary may only perform static analysis, meaning the adversary will never run the program or observe live execution, and their goal is to obtain the call graph of the obfuscated program. From Figure 1, it is apparent that with φ_0 being completely blank, the static adversary can only conduct intraprocedural analysis, as discussed in the previous section. While certain properties can be determined about $\mathcal{P}''$, even without a call graph, a knowledge of its true purpose is impossible with only intraprocedural analysis. The purpose of a program sometimes cannot even be inferred with the whole program understanding enabled by interprocedural analysis.

The only option left to the static analysis adversary is brute-forcing entries of Φ. Considering $\mathcal{P}''$ as seen in Figure 1, and that the adversary is leveraging no artifacts, if we know there are k call sites in Main() and i functions defined outside Main, then we know there are i^k possible programs. For example, with $i = 1$ function defined with $k = 30$ call sites, we would have 1^{30}, or only 1 possible program. Define S as the set of all semantically different programs that a program could execute. If we assume that Main does not have loops, and that no function calls any other function, then the size of S will be at most i^k. This upper bound exists because there will be some programs $P \in S$ that are semantically equivalent despite the functions executing in different orders.

The number of call sites and the number functions are not the only factors that decide the size of set S. Let us consider a program with k call sites and $i = 2$ functions. Function one will take in a list of numbers and output the same list, with every number shifted to the right with the last number being shifted around to the first. Function two will do the same operation except it will shift the numbers to the left. With our factors of $i = 2$ and k call sites, we can have at most a set S of size 2^k. $|S|$ could be quite large, but if when we consider all possible *semantically different* programs, we can see that the size of S will be quite small. In this example, this is because any combination of left shifts and right shifts can only create m different functionalities, where m is the size of the input list. The original exponential number of functionalities i^k will be capped at m.

5.1.3 Arbitrary Execution Analysis Adversary

We now consider the adversary that attempts to arbitrarily execute $\mathcal{P}''$ and dynamically analyze the running program. Protecting software with FT means that analysts must reverse engineer it dynamically because synchronization with $\mathcal{C}$ is needed for $\mathcal{P}''$ to operate at all (Figure 3). In other words, the author of $\mathcal{P}''$ would need to both authorize and expect any instance of the software running to coordinate the communication of $\mathcal{C}$ and $\mathcal{P}''$, as $\mathcal{C}$ could be executed on the equivalent of a license server. However, in the current implementation, there is nothing stopping arbitrary dynamic analyzers of $\mathcal{P}''$ from being able to start live-observation whenever they want. In future work, $\mathcal{C}$ could be improved to refuse to provide Flowtable updates if timing patterns indicate $\mathcal{P}''$ is being debugged or executed under other dynamic instrumentation.

5.1.4 Live-Observation Analysis Adversary

Finally, we consider the adversary who is prepared with introspective capabilities to observe $\mathcal{P}''$ when it traverses call graph edges stored in φ_n, as shown in Figure 3. FT does not effectively provide any absolute claims of protection against this adversary, although no obfuscation implementation currently *can*. In the worst case, the adversary can install hardware or use advanced instrumentation to observe all addresses that are called by `call` instructions. The adversary would still be limited by code coverage problems while trying to obtain the entire call graph of $\mathcal{P}''$.

5.2 Theoretical Efficiency

There are two sources of runtime cost:

Call indirection - FT makes every call more complicated by requiring a pointer dereference. This is measurable by the amount of extra instructions needed to perform the call. In LLVM IR every call becomes three instructions instead of one.

Flowtable (Φ) update latency - FT requires $\mathcal{P}''$ to get periodic flowtable updates, φ_n, in order to operate normally.

Consider the worst possible partitioning case for runtime performance: a partition for every function. In this case, we add $|exec(gate)|$ time to every call instruction. Since the gate function has constant execution time (it has no algorithmic dependencies on user input), we add constant time increase to every call instruction. We succeed in our initial goal of not increasing runtime by a non-polynomial factor, outlined in Section 3, even in the worst case. However, adding the order of 10's of instructions for every call instruction may not be practically feasible. Furthermore, though the gate function is not dependent on user input, it is a blocking synchronization function. Our testing implementation must wait for an IPC flowtable update before continuing. Even though these are expensive runtime costs,

in Section 5.3 we find that in the vast majority of cases, the runtime penalty is acceptable.

5.3 Empirical Results

5.3.1 Experimental Details

We used the LLVM test-suite framework to test FT. However, to build our Windows tool with the test-suite programs which were mostly targeted for a Linux platform, we used Cygwin. Even with Cygwin, only a subset of LLVM test-suite programs were compatible with Windows, but 58 programs still worked, which were sufficient for experimental tests. A list of the programs we tested is available in Appendix A, as well as timing data, size data, number of total call sites, number of transformed callsites and coverage.

By using the LLVM test-suite, we inherited the LLVM methodology for testing the correctness of their compiler and related tools. For each program in the LLVM test-suite that expects input, there are one or more reference input files. Each folder also contains each expected output for each input file. The testing framework makefiles compile each program with the native compiler and check its output for sanity. Then the makefiles compile the program with Clang and the Flowtables toolchain and run it in tandem with Flowcontrol to ensure that its output matches the native binary and reference output.

The tests were run on a Intel® Core© i7-3740QM CPU clocked at 2.70 GHz, 16 GB of RAM, running Windows 7 Enterprise 64-bit.

5.3.2 Results

The most important property of transforming $\mathcal{P}$ with FT is that the program functions exactly as it did before. After comparing the output of all $\mathcal{P}$ in our test suite and the output of all corresponding $\mathcal{P}''$ we found that for all 58 programs, the outputs matched and no illegal pointers from the flowtable were dereferenced. FT has 100% correctness for our test-suite.

Next we consider how much the implementation limitations listed in Section 4.2, actually affected our target programs. Table 1 presents the mean, standard deviation, minimum and maximum per program of total call sites ($|CS|$), those that could be transformed ($|CS'|$) and the numerical ratio between the two, termed *coverage* ($\frac{|CS'|}{|CS|}$).

	$\bar{x}$	σ	**Min**	**Max**
$\|CS\|$	29.017	37.109	2.000	260.000
$\|CS'\|$	7.189	7.834	1.000	41.000
$\frac{\|CS'\|}{\|CS\|}$	0.331	0.242	0.021	0.910

Table 1: Completeness Results Per Program

Lastly, we consider the performance of the test-suite programs after being transformed by FT. This is the key metric when determining the practicality of an obfuscation scheme. Our results show that FT is practical in a majority of cases, and with further investigation of partitioning schemes, may be practical for all cases.

The average multiplicative runtime increase (MRI) over all tests is 120x. On average the 58 $\mathcal{P}''$ run 120x slower than their $\mathcal{P}$ counterparts, but the corresponding standard deviation is 339x, which indicates wild deviations, especially since the theoretical lower limit for MRI is 1x. For the sake of figure clarity, in Figure 5a, we removed the twelve most deviant programs: `aha`, `bintr`, `bisort`, `health`, `mst`, `perimenter`, `treeadd`, `tsp`, `enc-md5`, `Towers`, `Treesort`, `sse.stepfft`, and were left with all programs with a MRI less than 25x. Figure 6b displays the MRI of every program in our test-suite.

We investigated why the test-suite programs had wildly deviant MRIs, and decided that the first viable explanation was that the programs with the highest coverage (as defined for Table 1) were the likeliest to have the greatest MRI. We interpreted the viability of this hypothesis with results in Figure 5b. But the data shows that there is no relationship between the coverage of our transformation in $\mathcal{P}''$ and how much $\mathcal{P}''$ slows down. Being inconclusive, we needed to continue our investigation of the outliers to find why they slowed down so greatly. We found that many partition boundaries in outlier $\mathcal{P}''$s fell in unfortunate locations, such as within tight loops. To determine whether this had a large impact on MRI, we conducted tests with programs that had 10 different random partitioning schemes each, as defined in Algorithm 1. In Figure 6, we compare the minimum, maximum and mean MRI of all 58 test programs run 10 different times with the same partition scheme (6a) to the same for the 58 test programs run with 10 different partitioning schemes (6b).

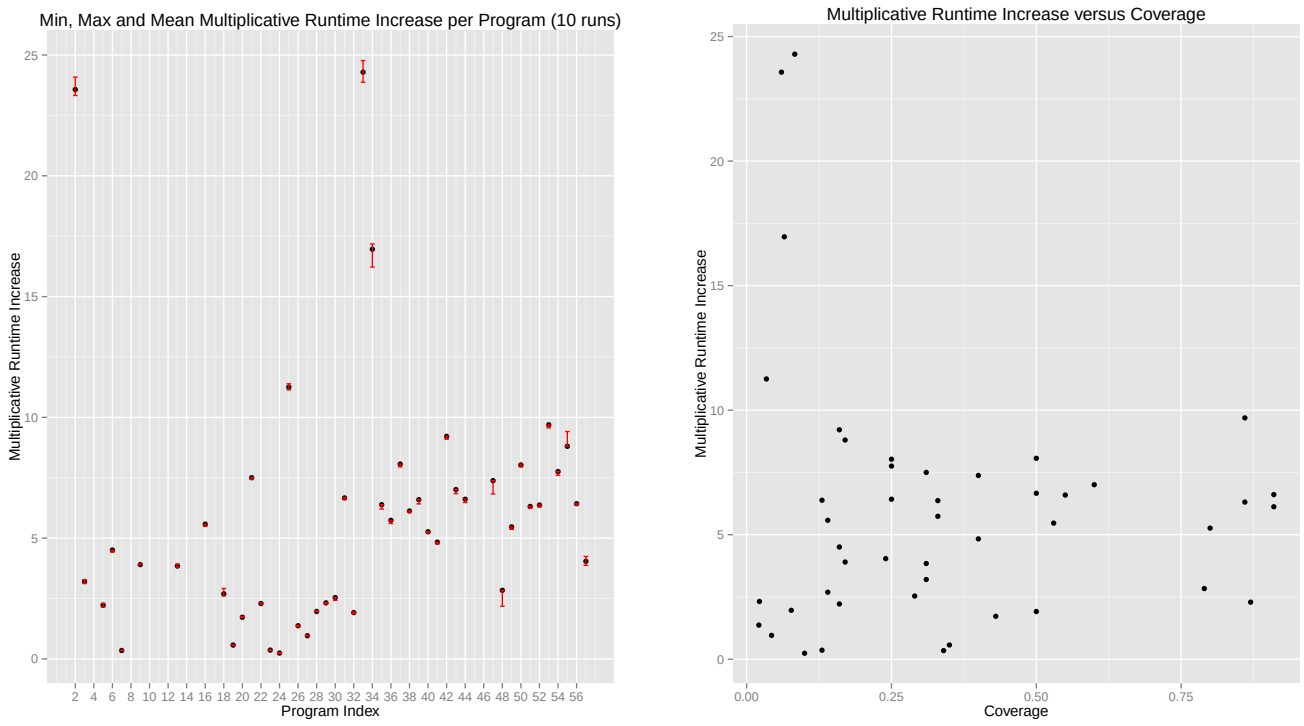
The results in Figure 6 clearly show that the partitioning scheme is easily the greatest factor in determining the MRI of $\mathcal{P}''$. Notably, these results are with essentially randomly different partitioning schemes. Even with unguided partitioning, `aha` (program 1) can be sped up by a third, `bintr` (program 4) can be sped up 18x, and `Treesort` (program 46) drops from 77x MRI to only 4x with a different partitioning scheme. Investigating intelligent partitioning more may yield even better results for performance.

6. RELATED WORK

In the realm of practical obfuscation, much attention has been focused on control flow graphs (CFGs) and the higher level call graphs. Many researchers have shown early attempts at control flow flattening to be vulnerable to attack [1]. Even commercial-grade obfuscation [7] has been shown to have significant weaknesses [25].

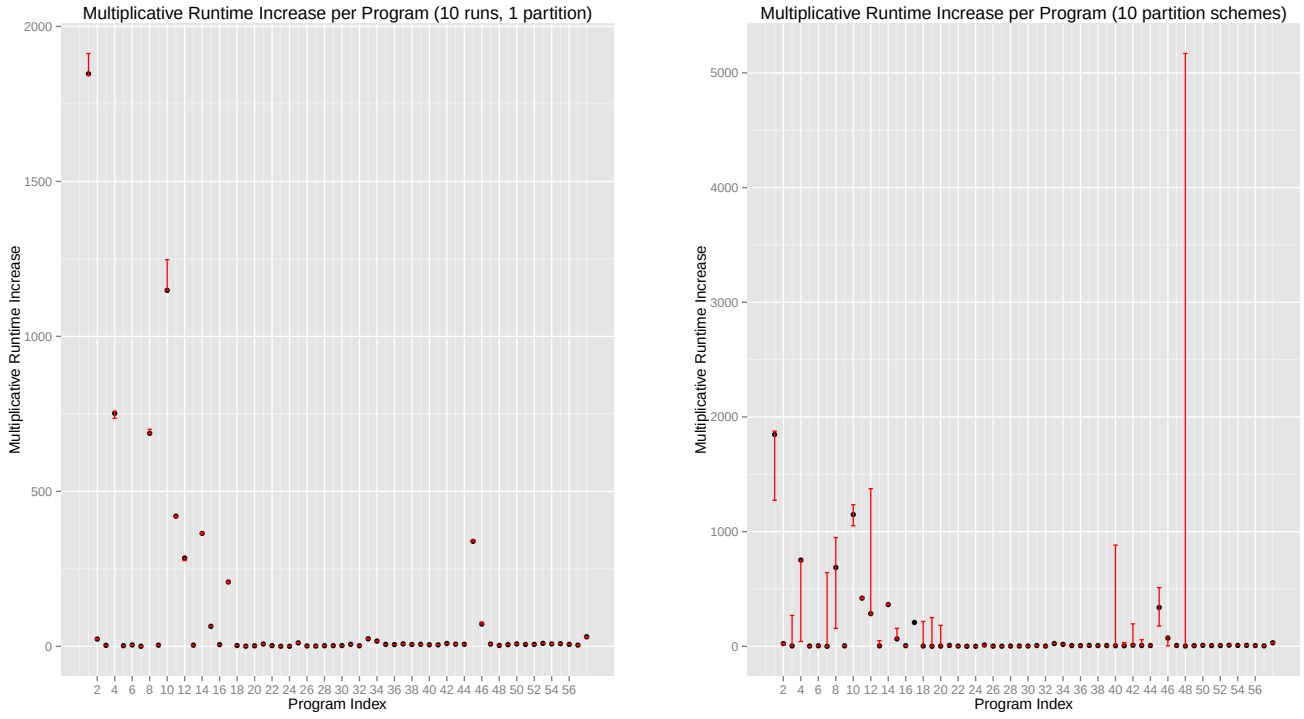
Practitioners of obfuscation soon realized the need for definable security properties of their schemes [10]. Some of the first attempts at this introduced NP-hard problems into various aspects of program analysis workflow [6]. Opaque predicates based on NP-Hard problems [19, 21, 27] provided a definable level of security for CFG and call graph protection schemes. Though these methods successfully and provably defeat a static analyzer, they do not protect against dynamic analysis. FT protects from arbitrary dynamic analysis by completely removing call graph edges from the obfuscated process.

The Nanomites technique [15] used by the Armadillo packer attempts to defeat static *and* dynamic analysis by binary packing and debug blocking, respectively. To use Nanomites, the code developer must annotate code segments that should be protected. All Armadillo protected binaries have a host process that starts a child process which the parent then starts debugging. The Windows' limitation of only allowing one attached debugger per process blocks other userland debuggers from attaching to the obfuscated binary. Next,



(a) (b)

Figure 5: MRI per program ($n_p = 4$) and MRI versus coverage for subset of test programs



(a) (b)

Figure 6: Comparing the range of MRI's for the same partition to 10 different partitions

to achieve debug blocking during unpacking, Armadillo replaces jumps in annotated code with 0xCC (int 3) so that when these software breakpoints are triggered, the host is responsible for replacing the 0xCC with an appropriate jump after cross-referencing three tables that contain the necessary data. Thus, the debugger is critical to the functioning of the protected code. FT and Nanomites are similar in that they remove the call graph from being normally resident in the target code. However, the methods diverge in two ways. First, FT's IR-level of analysis enables for more powerful transformations evident in the metamorphic potential, whereas the assembly-level that Nanomites operate on is much more restrictive. Also, FT is completely transparent to the code developer, but Nanomites requires source code annotation.

Royal suggests several methods in [23] that also force live-observation only, though by different means than our approach; namely, by providing no developer code on the host machine, but just an interpreter waiting for streamed instructions. That work did not implement an automatic transformation to a program of this form but rather offered an example of a current malware sample that exemplifies the proposed methods.

Little work has been published that is similar to FT's metamorphism. A a metamorphic engine for LLVM is explored in [24], but it only offers dead code insertion and subroutine permutation at compile-time.

7. FUTURE WORK AND CONCLUSION

There are several potential avenues of future work for Flowtables. First, further transformation granularities should be investigated, although they should be pursued realistically, as even function-level granularity sometimes causes prohibitive slowdown as we illustrate. Combining granularities would be a good solution for maintaining highest possible security of sensitive code, but allowing faster execution of less sensitive code. Extension of gating and partitioning to basic block granularity would be straightforward from function-level granularity. However, gating every instruction as we currently gate fringe functions will likely introduce prohibitive overhead into the system.

Another aspect of future work is related to optimizing the performance of Flowtables. Aspects of Flowtables that warrant further investigation for optimization are in both the partitioning and updating methodologies currently implemented. As noted in Section 5.3, partitioning can alter performance overheads drastically, so more intelligent approaches can focus on optimal program partitioning. Another possibility for reducing overhead would introduce code annotations to the programmer to indicate only the sensitive code portions that should be protected.

Additionally, there are several ways to increase the resilience of PSI against adversary analysis. Applying indirection to the arguments of functions would guarantee maximal search space for a static analysis adversary. We could possibly create another flowtable directing jumps to basic blocks with the push sequences preceding their functions, though due to LLVM's abstraction of function arguments this will not be straightforward. We could also apply indirection to library calls. A possibility would be to have the obfuscated program relay library information to its controller at start-up so that dynamically linked library calls can also be directed through the flowtable. Further research should pursue the notion of *Heisencode*, where code can be executed or observed, but one disturbs the other. We would pursue different means, extending PSI and the metamorphism that we have only begun to explore by creating a set of mutually dependent programs that have been linked together so that, through flowtable metamorphism, they may dynamically switch places with each other. This could enable protected code to be hidden with unprotected code and remain indiscernible.

Other future work may involve engineering efforts to turn Flowtables from a research prototype into a more mature solution for intellectual property protection. In the current development/testing state, Flowcontrol ($\mathcal{C}$) runs on the same system as $\mathcal{P}''$. Future work will enhance the security of $\mathcal{P}''$ by relocating $\mathcal{C}$ to a different system on the network that acts as a license server providing flowtable updates and other license verification features remotely, while also focusing on how to do this in the most efficient and effective way.

In general, software developers increasingly require intellectual property protection. Many obfuscation schemes in the past have offered only heuristic protection, though recently, researchers have published more schemes that blend theory and practice to have definable levels of security with an actual implementation and experimentation data. Program skeletal inversion transformations provided by Flowtables and its accompanying external metamorphism induction algorithm pragmatically precludes interprocedural analyses. This metamorphism used for IP protection has significant potential for extension to deeper security and potential integration with other LLVM tools.

8. REFERENCES

[1] B. Anckaert, M. Madou, B. De Sutter, B. De Bus, K. De Bosschere, and B. Preneel. Program obfuscation: A quantitative approach. In *Proceedings of the 2007 ACM Workshop on Quality of Protection*, QoP '07, pages 15–20, New York, NY, USA, 2007. ACM.

[2] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. *J. ACM*, 59(2):6:1–6:48, May 2012.

[3] L. Bassham and W. T. Polk. Threat assessment of malicious code and human threats. Technical report, National Institute of Standards and Technology, 1994. `http://csrc.nist.gov/publications/nistir/threats/subsubsection3_3_1_1.html`.

[4] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[5] B.-C. Cheng and W.-M. W. Hwu. Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 57–69, New York, NY, USA, 2000. ACM.

[6] S. Chow, Y. Gu, H. Johnson, and V. Zakharov. An approach to the obfuscation of control-flow of

sequential computer programs. In G. Davida and Y. Frankel, editors, *Information Security*, volume 2200 of *Lecture Notes in Computer Science*, pages 144–155. Springer Berlin Heidelberg, 2001.

[7] S. Chow, H. Johnson, and Y. Gu. Tamper resistant software-control flow encoding, Aug. 17 2004. US Patent 6,779,114.

[8] R. Chugh, J. W. Voung, R. Jhala, and S. Lerner. Dataflow analysis for concurrent programs using datarace detection. *SIGPLAN Not.*, 43(6):316–326, June 2008.

[9] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[10] L. D'Anna, B. Matt, A. Reisse, T. V. Vleck, S. Schwab, and P. Leblanc. Self-protecting mobile agents obfuscation report – final report. Technical report, NAI Labs, 2003. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.127.3668`.

[11] A. Dinaburg and A. Ruef. Mcsema: Static translation of x86 instructions to llvm. Technical report, Defense Advanced Research Project Agency, 2014. `https://www.trailofbits.com/resources/McSema.pdf`.

[12] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan. Repeatable reverse engineering for the greater good with panda. Technical report, Columbia University, 2014. `https://mice.cs.columbia.edu/getTechreport.php?techreportID=1588\&disposition=inline\&format=pdf`.

[13] M. Hlavac. stargazer: Latex code and ascii text for well-formatted regression and summary statistics tables. `https://cran.r-project.org/web/packages/stargazer/index.html`, 2014.

[14] W. Jin, C. Cohen, J. Gennari, C. Hines, S. Chaki, A. Gurfinkel, J. Havrilla, and P. Narasimhan. Recovering c++ objects from binaries using inter-procedural data-flow analysis. In *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*, PPREW'14, pages 1:1–1:11, New York, NY, USA, 2014. ACM.

[15] A. Kotik. Nanomite and debug blocker technologies: Scheme, pros, cons. `http://www.apriorit.com/white-papers/293-nanomite-technology`, 2013.

[16] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[17] O. Lhoták. Comparing call graphs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '07, pages 37–42, New York, NY, USA, 2007. ACM.

[18] J. Macdonald. On program security and obfuscation. `https://www.eecs.berkeley.edu/~daw/teaching/cs261-f98/projects/final-reports/jmacd.ps`, 1998.

[19] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 421 –430, dec. 2007.

[20] C. L. Noll, J. Horn, P. Seebach, and L. A. Broukhis. International obfuscated c code contest. `http://www.ioccc.org/`.

[21] T. Ogiso, Y. Sakabe, M. Soshi, and A. Miyaji. Software tamper resistance based on the difficulty of interprocedural analysis. In *of Interprocedural Analysis, 3 rd International Workshop on Information Security Applications*, pages 437–452, 2002.

[22] M. Schiffman. A brief history of malware obfuscation. `https://blogs.cisco.com/security/a_brief_history_of_malware_obfuscation_part_1_of_2`, February 2010. Cisco.

[23] C. Song and P. Royal. Flowers for automated malware analysis. Technical report, Georgia Institute of Technology, 2012. `https://media.blackhat.com/bh-us-12/Briefings/Song/BH_US_12_Song_Royal_Flowers_Automated_WP.pdf`.

[24] T. Tamboli, T. Austin, and M. Stamp. Metamorphic code generation from llvm bytecode. *Journal of Computer Virology and Hacking Techniques*, 10(3):177–187, 2014.

[25] S. Udupa, S. Debray, and M. Madou. Deobfuscation: reverse engineering obfuscated code. In *Reverse Engineering, 12th Working Conference on*, pages 10 pp.–, Nov 2005.

[26] A. Venkatesan. Code obfuscation and virus detection. Master's thesis, San Jose State University, May 2008. `http://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=1115\&context=etd_projects`.

[27] C. Wang, J. Hill, J. Knight, and J. Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical report, University of Virginia, Charlottesville, VA, USA, 2000. `http://www.ncstrl.org:8900/ncstrl/servlet/search?formname=detail\&id=oai\%3Ancstrlh\%3Auva_cs\%3AUVA_CS\%2F\%2FCS-2000-12`.

[28] W. Wong and M. Stamp. Hunting for metamorphic engines. *Journal in Computer Virology*, 2(3):211–229, 2006.

APPENDIX

A. RAW DATA

Table 2

	Names	Before(s)	After(s)	Before(B)	After(B)	$\|CS\|$	$\|CS'\|$	Coverage
1	aha	1.621	2,995.686	13900	17372	32	9	0.280
2	CrystalMk	4.931	121.170	12500	16200	50	3	0.060
3	iotest	0.308	1.294	7816	11320	13	4	0.310
4	bintr	0.222	167.141	7972	11552	57	18	0.320
5	automotive-basicmath	6.252	20.132	8184	11796	56	9	0.160
6	security-blowfish	0.172	0.947	46556	50260	260	41	0.160
7	is	5.592	7.535	9448	13048	59	20	0.340
8	bisort	0.548	377.196	7788	11384	31	13	0.420
9	em3d	2.664	13.062	17364	21140	96	16	0.170
10	health	0.390	448.327	18748	22488	56	16	0.290
11	mst	0.183	77.029	10944	14512	48	6	0.130
12	perimeter	0.363	103.822	7956	11576	37	14	0.380
13	power	0.977	4.732	16136	19800	75	23	0.310
14	treeadd	1.561	570.171	3788	7376	16	6	0.380
15	tsp	0.921	60.487	14256	18012	60	9	0.150
16	allroots	0.142	0.934	8324	12000	50	7	0.140
17	enc-md5	1.236	257.953	17364	20968	32	6	0.190
18	enc-pc1	0.638	2.355	11372	14912	7	1	0.140
19	ecbdes	1.909	3.001	28476	32020	31	11	0.350
20	n-body	0.796	2.167	5492	9116	7	3	0.430
21	puzzle	0.246	2.093	5424	8912	16	5	0.310
22	spectral-norm	0.733	2.414	5272	8808	23	20	0.870
23	dry	1.149	1.567	5632	9184	8	1	0.130
24	fldry	1.259	1.562	6116	9664	10	1	0.100
25	misr	0.297	3.639	9528	13156	29	1	0.034
26	himenobmtxpa	0.817	1.938	18128	21672	47	1	0.021
27	matmul_f64_4x4	1.358	2.659	5680	9224	23	1	0.043
28	ReedSolomon	3.552	10.528	19032	22776	26	2	0.077
29	richards_benchmark	0.742	2.463	13164	16852	46	1	0.022
30	salsa20	5.299	18.748	5504	9060	7	2	0.290
31	ackermann	0.122	0.935	2284	5784	4	2	0.500
32	fib2	1.294	3.774	2232	5736	4	2	0.500
33	hash	4.025	101.798	6040	9632	24	2	0.083
34	lists	4.993	89.678	10040	13640	31	2	0.065
35	objinst	0.128	0.945	4120	7680	32	4	0.130
36	Bubblesort	0.149	1.004	3524	7052	3	1	0.330
37	FloatMM	0.210	1.904	4144	7676	2	1	0.500
38	IntMM	0.130	0.926	4020	7544	11	10	0.910
39	Oscar	0.125	0.949	6624	10188	22	12	0.550
40	Perm	0.152	0.952	3456	6972	10	8	0.800
41	Puzzle	0.215	1.254	11060	14600	10	4	0.400
42	Queens	0.094	0.960	4636	8148	25	4	0.160
43	Quicksort	0.126	1.009	3928	7448	5	3	0.600
44	RealMM	0.123	0.936	4104	7632	11	10	0.910
45	Towers	0.152	51.652	6348	9868	19	5	0.260
46	Treesort	0.223	16.267	5320	8852	18	4	0.220
47	compare	0.111	0.930	1828	5312	5	2	0.400
48	uint64_to_float	0.527	2.023	5036	8624	38	30	0.790
49	bigstack	0.144	0.931	5448	9012	19	10	0.530
50	2011-03-28-Bitfield	0.102	0.921	1596	5120	4	1	0.250
51	2008-04-18-LoopBug	0.126	0.921	2232	5728	7	6	0.860
52	2009-12-07-StructReturn	0.125	0.921	2232	5764	6	2	0.330
53	2008-04-20-LoopBug2	0.087	0.930	2180	5680	7	6	0.860
54	byval-alignment	0.106	0.928	2028	5552	4	1	0.250
55	2003-07-09-SignedArgs	0.090	0.881	2860	6380	12	2	0.170
56	2007-03-02-VaCopy	0.125	0.928	1536	5040	4	1	0.250
57	sse.expandfft	0.481	2.424	10128	13712	17	4	0.240
58	sse.stepfft	0.495	15.802	7932	11544	21	8	0.380